[Post Session Activity for Students](https://aispry.com/theme/university/pages/forum_answer.php?questionid=1597)

Python:

1]Problem Statement:

You are given a collection of intervals, where each interval is represented as a pair of integers `[start, end]`. Your task is to write a Python function to merge overlapping intervals and return a new collection of non-overlapping intervals.

For example, given the input:

```

intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]

```

The function should return:

```

[[1, 6], [8, 10], [15, 18]]

```

Explanation:

- The intervals `[1, 3]` and `[2, 6]` overlap, so they should be merged into `[1, 6]`.

- The intervals `[8, 10]` and `[15, 18]` do not overlap with any other interval, so they remain unchanged.

Algorithm:

1. Sort the intervals based on their start values.

2. Initialize an empty list to store the merged intervals.

3. Iterate through the sorted intervals:

   - If the current interval overlaps with the previous merged interval or if it can be merged with the last merged interval, update the last merged interval's end value.

   - Otherwise, add the current interval to the merged list.

4. Return the merged list of intervals.

def merge\_intervals(intervals):

if not intervals:

return []

intervals.sort(key=lambda x: x[0]) # Sort intervals based on start values

merged = [intervals[0]] # Initialize merged list with the first interval

for interval in intervals[1:]:

if interval[0] <= merged[-1][1]: # Check for overlap

merged[-1][1] = max(merged[-1][1], interval[1]) # Merge intervals

else:

merged.append(interval) # Add non-overlapping interval

return merged

# Test the function

intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]

print(merge\_intervals(intervals))

**Output:**

[[1, 6], [8, 10], [15, 18]]

2]

Question:

Given a set of items, each with a weight and a value, determine the maximum value that can be obtained by selecting a subset of the items such that the sum of the weights of the selected items is less than or equal to a given weight limit.

Example:

Suppose we have the following items:

- Item 1: Weight = 2, Value = 3

- Item 2: Weight = 3, Value = 4

- Item 3: Weight = 4, Value = 5

- Item 4: Weight = 5, Value = 6

Weight limit: 8

Output:

The maximum value that can be obtained by selecting items while ensuring that the total weight does not exceed the weight limit.

Input:

Items:

1. (2, 3)

2. (3, 4)

3. (4, 5)

4. (5, 6)

Weight Limit: 8

Output:

Maximum value: 10

Explanation:

By selecting Item 1 and Item 2 (total weight = 2 + 3 = 5, total value = 3 + 4 = 7), we get the maximum value of 7, which does not exceed the weight limit of 8.

def knapsack(items, weight\_limit):

max\_value = {0: 0}

for weight, value in items:

for w in range(weight\_limit, weight - 1, -1):

if w - weight in max\_value:

max\_value[w] = max(max\_value.get(w, 0), max\_value[w - weight] + value)

return max\_value[weight\_limit]

# Test the function

items = [(2, 3), (3, 4), (4, 5), (5, 6)]

weight\_limit = 8

print("Maximum value:", knapsack(items, weight\_limit))

**Output:**

Maximum value: 10

3]

Question:

Implement the Tower of Hanoi problem using recursion. Given three rods and a number of disks of different sizes, the objective is to move the entire stack to another rod, obeying the following rules:

1. Only one disk can be moved at a time.

2. Each move consists of taking the top disk from one stack and placing it on another stack.

3. No disk may be placed on top of a smaller disk.

Example:

Suppose we have 3 disks initially placed on the first rod (source rod) and we want to move them to the third rod (destination rod).

Input:

Number of disks: 3

Source rod: A

Auxiliary rod: B

Destination rod: C

Output:

The sequence of steps required to move the disks from the source rod to the destination rod following the Tower of Hanoi rules.

Explanation:

For 3 disks, the steps would be as follows:

1. Move disk 1 from rod A to rod C.

2. Move disk 2 from rod A to rod B.

3. Move disk 1 from rod C to rod B.

4. Move disk 3 from rod A to rod C.

5. Move disk 1 from rod B to rod A.

6. Move disk 2 from rod B to rod C.

7. Move disk 1 from rod A to rod C.

Thus, the sequence of steps to move the 3 disks from rod A to rod C following Tower of Hanoi rules is provided as output.

def tower\_of\_hanoi(n, source, auxiliary, destination):

if n == 1:

print("Move disk 1 from rod", source, "to rod", destination)

return

tower\_of\_hanoi(n-1, source, destination, auxiliary)

print("Move disk", n, "from rod", source, "to rod", destination)

tower\_of\_hanoi(n-1, auxiliary, source, destination)

# Test the function

num\_disks = 3

source\_rod = 'A'

auxiliary\_rod = 'B'

destination\_rod = 'C'

print("Sequence of steps to move", num\_disks, "disks:")

tower\_of\_hanoi(num\_disks, source\_rod, auxiliary\_rod, destination\_rod)

**Output:**

Sequence of steps to move 3 disks:

Move disk 1 from rod A to rod C

Move disk 2 from rod A to rod B

Move disk 1 from rod C to rod B

Move disk 3 from rod A to rod C

Move disk 1 from rod B to rod A

Move disk 2 from rod B to rod C

Move disk 1 from rod A to rod C

4]

Question:

Write a program to check if a given string of parentheses is balanced.

Sample Input:

String: "((()))"

String: "(()())"

String: "((())"

String: "()())"

Sample Output:

For the input "((()))", the output is "Balanced".

For the input "(()())", the output is "Balanced".

For the input "((())", the output is "Not Balanced".

For the input "()())", the output is "Not Balanced".

Explanation:

A string of parentheses is considered balanced if every opening parenthesis has a corresponding closing parenthesis and they occur in the correct order. To check for balance, we can use a stack data structure. We iterate through the string and push each opening parenthesis onto the stack. When we encounter a closing parenthesis, we pop the top element from the stack and check if it matches the corresponding opening parenthesis. If it matches, we continue; otherwise, the parentheses are not balanced.

def is\_balanced(parentheses):

stack = []

for char in parentheses:

if char == '(':

stack.append(char)

elif char == ')':

if not stack or stack.pop() != '(':

return False

return len(stack) == 0

# Test the function

inputs = ["((()))", "(()())", "((())", "()())"]

for input\_str in inputs:

if is\_balanced(input\_str):

print(f'For the input "{input\_str}", the output is "Balanced".')

else:

print(f'For the input "{input\_str}", the output is "Not Balanced".')

**Output:**

For the input "((()))", the output is "Balanced".

For the input "(()())", the output is "Balanced".

For the input "((())", the output is "Not Balanced".

For the input "()())", the output is "Not Balanced".

SQL:

1] Find the Customer Names in the Central Region with Ages between 30 and 40:

SELECT CustomerName

FROM Customers

WHERE Region = 'Central' AND Age BETWEEN 30 AND 40;

2] List the Customer Names and their Ages from the East Region, sorted by Age in descending order:

SELECT CustomerName, Age

FROM Customers

WHERE Region = 'East'

ORDER BY Age DESC;

3] List the Customer Names and their Segment Names who are from Cities in the 'Central' Region sorted by Age in descending order:

SELECT c.CustomerName, s.SegmentName

FROM Customers c

JOIN Segments s ON c.SegmentID = s.SegmentID

WHERE c.Region = 'Central'

ORDER BY c.Age DESC;